Overview

° Word/ Halfword/ Byte Addressing
° Byte ordering
° Signed Load Instructions
° Instruction Support for Characters

Review: Assembly Operands: Memory

° C variables map onto registers; what about large data structures like arrays?
° 1 of 5 components of a computer: memory contains such data structures
° But ARM arithmetic instructions only operate on registers, never directly on memory.

Data transfer instructions transfer data between registers and memory:
  • Memory to register
  • Register to memory

Review: Data Transfer: Memory ↔ Reg

° Example: ldr a1, [v1, #8] Similar instructions
° Example: ldr a1, [v1, v2] For STR
° Example: ldr a1, [v1,#12]!
  Pre Indexed Load: Subsequently, v1 is updated by the computed sum of v1 and 12, (v1 ← v1 + 12).
° Example: ldr a1, [v1, v2]!
  Pre Indexed Load: Subsequently, v1 is updated by the computed sum of v1 and v2, (v1 ← v1 + v2).
° Example: ldr a1, [v1],#12
  Post Indexed Load: Subsequently, v1 is updated by the computed sum of v1 and 12, (v1 ← v1 + 12).
° Example: ldr a1, [v1], v2
  Post Indexed Load: Subsequently, v1 is updated by the computed sum of v1 and v2, (v1 ← v1 + v2).
Review: Memory Alignment

ARM requires that all words start at addresses that are multiples of 4 bytes.

- Called Alignment: objects must fall on address that is multiple of their size.
- Some machines like Intel allow non-aligned accesses.

Data Transfer: More Mem to Reg Variants (#1/2)

- Load Byte Example:
  ```
  ldrb a1, [v1,#12]
  ```
  This instruction will take the pointer in `v1`, add 12 bytes to it, and then load the byte value from the memory pointed to by this calculated sum into register `a1`.

- Load Byte Example:
  ```
  ldrb a1, [v1, v2]
  ```
  This instruction will take the pointer in `v1`, add an index offset in register `v2` to it, and then load the byte value from the memory pointed to by this calculated sum into register `a1`.

Data Transfer: More Reg to Mem Variants (#1/2)

- Store Byte Example:
  ```
  strb a1, [v1,#12]
  ```
  This instruction will take the pointer in `v1`, add 12 bytes to it, and then store the value from the least significant byte (lsb) of register `a1` into the memory address pointed to by the calculated sum.

- Store Byte Example:
  ```
  strb a1, [v1, v2]
  ```
  This instruction will take the pointer in `v1`, adds register `v2` to it, and then store the value from the lsb of register `a1` into the memory address pointed to by the calculated sum.

1 word = 4 Bytes
Data Transfer: More Reg to Mem Variants (#2/2)

° Store Half Word Example:

\texttt{strh a1, [v1,#12]}

This instruction will take the pointer in \texttt{v1}, add 12 bytes to it, and then store the value from \texttt{half word} of register \texttt{a1} into the memory address pointed to by the calculated sum.

° Store Half Word Example:

\texttt{strh a1,[v1, v2]}

This instruction will take the pointer in \texttt{v1}, adds register \texttt{v2} to it, and then store the value from \texttt{half word} of register \texttt{a1} into the memory address pointed to by the calculated sum.

Compilation with Memory (Byte Addressing)

° What offset in \texttt{ldr} to select \texttt{my_Array[8]} (defined as \texttt{Char}) in C?

° 1x8=8 to select \texttt{my_Array[8]}: byte

° Compile by hand using registers:

\begin{equation*}
g = h + \texttt{my_Array[8]};
\end{equation*}

\begin{itemize}
\item \texttt{g}: v1, h: v2, v3: base address of \texttt{my_Array}
\end{itemize}

° 1st transfer from memory to register:

\begin{equation*}
\texttt{ldrb v1, [v3,#8] ; v1 gets my_Array[8]}
\end{equation*}

\begin{itemize}
\item Add 8 to r3 to select \texttt{my_Array[8]}, put into \texttt{v1}
\end{itemize}

° Next add it to \texttt{h} and place in \texttt{g}

\begin{equation*}
\texttt{add v1,v2,v1 ; v1 = h+ my_Array[8]}
\end{equation*}

Compilation with Memory (Half Word Addressing)

° What offset in \texttt{ldr} to select \texttt{my_Array[8]} (defined as \texttt{halfword}) in C?

° 2x8=16 to select \texttt{my_Array[8]}: byte

° Compile by hand using registers:

\begin{equation*}
g = h + \texttt{my_Array[8]};
\end{equation*}

\begin{itemize}
\item \texttt{g}: v1, h: v2, v3: base address of \texttt{my_Array}
\end{itemize}

° 1st transfer from memory to register:

\begin{equation*}
\texttt{ldrh v1, [v3, #16] ; v1 gets my_Array[8]}
\end{equation*}

\begin{itemize}
\item Add 16 to r3 to select \texttt{my_Array[8]}, put into \texttt{v1}
\end{itemize}

° Next add it to \texttt{h} and place in \texttt{g}

\begin{equation*}
\texttt{add v1,v2,v1 ; v1 = h+ my_Array[8]}
\end{equation*}

More Notes about Memory: Word

° How are bytes numbered in a word?

\begin{itemize}
\item Little Endian byte 0
\item Big Endian byte 0
\end{itemize}

° Gulliver’s Travels: Which end of egg to open?


° Little Endian address of least significant byte: Intel 80x86, DEC Alpha,

° Big Endian address of most significant byte

HP PA, IBM/Motorola PowerPC, SGI, Sparc

° ARM is Little Endian by default, However it can be made Big Endian by configuration.
**Endianness Example**

- Little-endian:
  - r0 = 0x11223344
  - Memory:
    - r1 = 0x100
    - r2 = 0x44

- Big-endian:
  - r1 = 0x100
  - r2 = 0x11

**Code Example**

- Write a segment of code that add together elements x to x+(n-1) of an array, where the element x = 0 is the first element of the array.
- Each element of the array is word sized (ie. 32 bits).
- The segment should use post-indexed addressing.
- At the start of your segments, you should assume that:
  - a1 points to the start of the array.
  - a2 = x
  - a3 = n

```
° ARM instruction (ldrsb) automatically extends “sign” of byte for load byte.
    ldrsh a1, [v1,#12] ldrsh a1, [v1,v2]
° ARM instruction (ldrsh) automatically
    extends “sign” of half word for load half word.
    ldrsh a1, [v1,#12] ldrsh a1, [v1,v2]
```
Instruction Support for Characters

- ARM (and most other instruction sets) include instructions to operate on bytes:
  - move byte (ldrb) loads a byte from memory/reg, placing it in rightmost 8 bits of a register, or vice versa

- Declares byte variables in C as “char”

- Assume x, y are declared char. x in memory at [v1,#4] and y at [v1,#0].
  What is ARM code for x = y;

```asm
ldrb a1, [v1,#0]
strb a1, [v1,#4] ; transfer y to x
```

Strings in C: Example

- String simply an array of char

```c
void strcpy (char x[], char y[]){
  int i = 0; /* declare, initialize i*/
  while ((x[i] = y[i]) != '\0') /* 0 */
    i = i + 1; /* copy and test byte */
}
```

- function

```asm
i, addr of x[0], addr of y[0]: v1, a1, a2
func ret addr: lr
```

```asm
strcpy:
  mov v1, #-1             ; i = -1
  L1: add v1, v1, #1        ; i = i + 1
  ldrb a3, [a2,v1] ; a1= y[i]
  strb a3, [a1,v1]        ; x[i]=y[i]
  cmp a3, #0
  bne L1 ; y[i]!=0
goto L1
  mov pc, lr ; return
```

- ideally compiler optimizes code for you

Strings in C: Example using pointers

- String simply an array of char

```c
void strcpy2 (char *px, char *py){
  while ((*px++ = *py++) != '\0') /* 0 */
    ; /* copy and test byte */
}
```

- function

```asm
addr of x[0], addr of y[0]: v2, v3
func ret addr: lr
```

```asm
strcpy:
  L1: ldrb a1, [v3],#1     ;a1= *py, py = py +1
  strb a1, [v2],#1 ; px = px +1
  cmp a1, #0
  bne L1 ; py!=0 goto L1
  mov pc, lr ; return
```

Block Copy Transfer (#1/5)

- Consider the following code:

```asm
str a1, [v1],#4
str a2, [v1],#4
str a3, [v1],#4
str a4, [v1],#4
```

Replace this with
```
stmia v1!, {a1-a4}
```

- STMIA : SORE MULTIPLE INCREMENT AFTER

- Consider the following code:

```asm
str a1, [v1, #4]!
str a2, [v1, #4]!
str a3, [v1, #4]!
str a4, [v1, #4]!
```

Replace this with
```
stmib v1!, {a1-a4}
```

- STMB : SORE MULTIPLE INCREMENT BEFORE
Block Copy Transfer (#2/5)

° Consider the following code:
  
  ```
  str a1, [v1], #-4  
  str a2, [v1], #-4  
  str a3, [v1], #-4  
  str a4, [v1], #-4  
  ```

  Replace this with
  ```
  stmda v1!, {a1-a4}  
  ```

  **STMAD**: SORE MULTIPLE DECREMENT AFTER

° Consider the following code:
  
  ```
  str a1, [v1, #-4]!  
  str a2, [v1, #-4]!  
  str a3, [v1, #-4]!  
  str a4, [v1, #-4]!  
  ```

  Replace this with
  ```
  stmdb v1!, {a1-a4}  
  ```

  **STMDB**: SORE MULTIPLE DECREMENT BEFORE

Block Copy Transfer (#3/5)

° Consider the following code:
  
  ```
  str a1, [v1]  
  str a2, [v1,#4]  
  str a3, [v1,#8]  
  str a4, [v1,#12]  
  ```

  Replace this with
  ```
  stmia v1, {a1-a4}  
  ```

  **STMIA**: SORE MULTIPLE INCREMENT AFTER

° Consider the following code:
  
  ```
  str a1, [v1, #4]  
  str a2, [v1, #8]  
  str a3, [v1, #12]  
  str a4, [v1, #16]  
  ```

  Replace this with
  ```
  stmib v1, {a1-a4}  
  ```

  **STMIB**: SORE MULTIPLE INCREMENT BEFORE

Block Copy Transfer (#4/5)

° Consider the following code:
  
  ```
  str a1, [v1]  
  str a2, [v1,#-4]  
  str a3, [v1,#-8]  
  str a4, [v1,#-12]  
  ```

  Replace this with
  ```
  stmda v1, {a1-a4}  
  ```

  **STMAD**: SORE MULTIPLE DECREMENT AFTER

° Consider the following code:
  
  ```
  str a2, [v1,#-4]  
  str a3, [v1,#-8]  
  str a4, [v1,#-12]  
  str a1, [v1,#16]  
  ```

  Replace this with
  ```
  stmdb v1, {a1-a4,}  
  ```

  **STMDB**: SORE MULTIPLE DECREMENT BEFORE

Block Data Transfer (#5/5)

° Similarly we have
  
  - **LDMIA**: Load Multiple Increment After
  - **LDMIB**: Load Multiple Increment Before
  - **LDMDA**: Load Multiple Decrement After
  - **LDMDB**: Load Multiple Decrement Before

  For details See Chapter 3, page 61 – 62
  
“And in Conclusion…” (#1/2)

- In ARM Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster

- Memory is byte-addressable, but \texttt{ldr} and \texttt{str} access one word at a time.

- Access byte and halfword using \texttt{ldrb}, \texttt{ldrh}, \texttt{ldrsb} and \texttt{ldrsh}

- A pointer (used by \texttt{ldr} and \texttt{str}) is just a memory address, so we can add to it or subtract from it (using offset).

“And in Conclusion…” (#2/2)

- New Instructions:
  - \texttt{ldr}, \texttt{str}
  - \texttt{ldrb}, \texttt{strb}
  - \texttt{ldrh}, \texttt{strh}
  - \texttt{ldrsb}, \texttt{ldrsh}